Week 11.1

2303A51018

B-28

Task Description #1 – Stack Implementation

Task: Use AI to generate a Stack class with push, pop, peek, and is_empty methods.

Sample Input Code:

```
class Stack:
    pass
```

Expected Output:

- A functional stack implementation with all required methods and docstrings.

Code

Thonny - C:\Users\HP\Downloads\AI Ass 11.1.py @ 48 : 37

File Edit View Run Device Tools Help

AI Ass 11.1.py

```python
class Stack:
    """
    A simple implementation of a stack data structure (LIFO - Last In, First Out).
    """

    def __init__(self):
        """
        Initialize an empty stack.
        """
        self._items = []

    def push(self, item):
        """
        Add an item to the top of the stack.

        :param item: The item to be added to the stack.
        """
        self._items.append(item)

    def pop(self):
        """
        Remove and return the top item of the stack.

        :return: The top item of the stack.
        :raises IndexError: If the stack is empty.
        """
        if self.is_empty():
            raise IndexError("pop from empty stack")
        return self._items.pop()

    def peek(self):
        """
        Return the top item of the stack without removing it.

        :return: The top item of the stack.
        :raises IndexError: If the stack is empty.
        """
        if self.is_empty():
            raise IndexError("peek from empty stack")
        return self._items[-1]

    def is_empty(self):
        """
        Check whether the stack is empty.

        :return: True if the stack is empty, False otherwise.
        """
        return len(self._items) == 0
```

Output
True
30
30
20
False

Explanation

A **Stack** follows the **LIFO (Last In, First Out)** principle. This means the last element added to the stack is the first one to be removed.

Let's walk through the example step-by-step:

1. **`s = Stack()`**
    a. A new empty stack is created.
    b. Internally, `_items = []`.
2. **`print(s.is_empty())` → `True`**
    a. The stack is empty because no elements have been added yet.
3. **`s.push(10)`**
    a. Stack becomes: `[10]`
4. **`s.push(20)`**
    a. Stack becomes: `[10, 20]`
5. **`s.push(30)`**
    a. Stack becomes: `[10, 20, 30]`
6. **`print(s.peek())` → `30`**
    a. `peek()` returns the top element **without removing it**.
    b. Stack remains: `[10, 20, 30]`
7. **`print(s.pop())` → `30`**
    a. `pop()` removes and returns the top element.
    b. Stack becomes: `[10, 20]`
8. **`print(s.pop())` → `20`**
    a. Removes and returns the new top element.
    b. Stack becomes: `[10]`
9. **`print(s.is_empty())` → `False`**
    a. The stack still contains `[10]`, so it is not empty.

📌 Conclusion

- The stack works using the **Last In, First Out (LIFO)** principle.
- `push()` adds elements to the top.
- `pop()` removes the most recently added element.
- `peek()` views the top element without removing it.
- `is_empty()` checks whether the stack has any elements.

This implementation correctly demonstrates how a stack behaves and how its core operations function.

Task Description #2 – Queue Implementation

Task: Use AI to implement a Queue using Python lists.

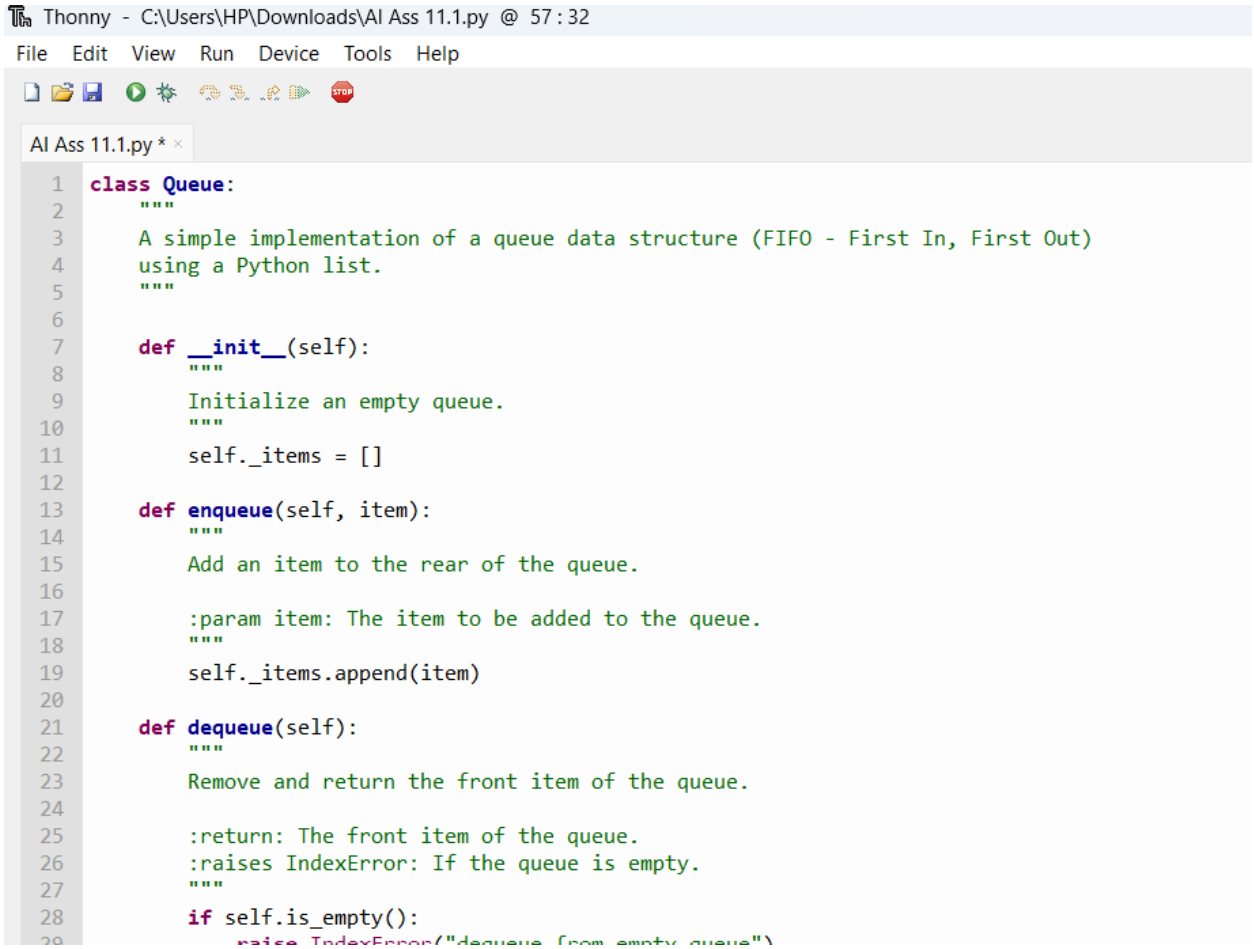
Sample Input Code:

```
class Queue:
    pass
```

Expected Output:

- FIFO-based queue class with enqueue, dequeue, peek, and size methods.

Code


Thonny - C:\Users\HP\Downloads\AI Ass 11.1.py @ 57 : 32
File Edit View Run Device Tools Help
AI Ass 11.1.py *
class Queue:
"""
A simple implementation of a queue data structure (FIFO - First In, First Out)
using a Python list.
"""
def __init__(self):
"""
Initialize an empty queue.
"""
self._items = []
def enqueue(self, item):
"""
Add an item to the rear of the queue.
:param item: The item to be added to the queue.
"""
self._items.append(item)
def dequeue(self):
"""
Remove and return the front item of the queue.
:return: The front item of the queue.
:raises IndexError: If the queue is empty.
"""
if self.is_empty():

```
def peek(self):
    """
    Return the front item without removing it.

    :return: The front item of the queue.
    :raises IndexError: If the queue is empty.
    """
    if self.is_empty():
        raise IndexError("peek from empty queue")
    return self._items[0]

def is_empty(self):
    """
    Check whether the queue is empty.

    :return: True if the queue is empty, False otherwise.
    """
    return len(self._items) == 0

def size(self):
    """
    Return the number of items in the queue.

    :return: The size of the queue.
    """
    return len(self._items)
```

Output

```
True
10
10
2
False
```

Explanation

A **Queue** follows the **FIFO (First In, First Out)** principle.
This means the first element added to the queue is the first one to be removed.

Let's go step by step through the example:

1. **`q = Queue()`**
   a. A new empty queue is created.
   b. Internally: `_items = []`
2. **`print(q.is_empty())` → `True`**
   a. The queue is empty because no elements have been added yet.
3. **`q.enqueue(10)`**
   a. Queue becomes: `[10]`
4. **`q.enqueue(20)`**
   a. Queue becomes: `[10, 20]`

5. **`q.enqueue(30)`**
    a. Queue becomes: `[10, 20, 30]`
6. **`print(q.peek())` → `10`**
    a. `peek()` returns the **front element** without removing it.
    b. Queue remains: `[10, 20, 30]`
7. **`print(q.dequeue())` → `10`**
    a. `dequeue()` removes and returns the **first element**.
    b. Queue becomes: `[20, 30]`
8. **`print(q.size())` → `2`**
    a. Two elements remain in the queue.
9. **`print(q.is_empty())` → `False`**
    a. The queue still contains elements, so it is not empty.

📌 Conclusion

- A queue works on the **First In, First Out (FIFO)** principle.
- `enqueue()` adds elements to the **rear**.
- `dequeue()` removes elements from the **front**.
- `peek()` views the front element without removing it.
- `size()` returns the number of elements.
- `is_empty()` checks whether the queue contains any items.

This implementation correctly demonstrates how a queue operates using Python lists.

Task Description #3 – Linked List

Task: Use AI to generate a Singly Linked List with insert and display methods.

Sample Input Code:

```
class Node:
    pass

class LinkedList:
    pass
```

Expected Output:

- A working linked list implementation with clear method documentation.

Code

```
class Node:
    """
    A node in a singly linked list.

    Attributes:
        data: The value stored in the node.
        next: Reference to the next node in the list.
    """

    def __init__(self, data):
        """
        Initialize a node with data and set next to None.

        :param data: The value to store in the node.
        """
        self.data = data
        self.next = None


class LinkedList:
    """
    A simple implementation of a singly linked list.

    Methods:
        insert(data): Insert a new node at the end of the list.
        display(): Print all elements in the list.
    """
```

```
def __init__(self):
    """
    Initialize an empty linked list.
    """
    self.head = None

def insert(self, data):
    """
    Insert a new node with the given data at the end of the list.

    :param data: The value to insert into the list.
    """
    new_node = Node(data)

    # If the list is empty, make the new node the head
    if self.head is None:
        self.head = new_node
        return

    # Otherwise, traverse to the end and append the new node
    current = self.head
    while current.next:
        current = current.next

    current.next = new_node

def display(self):
    """
    Display all elements of the linked list.
```

```
def display(self):
    """
    Display all elements of the linked list.
    """
    current = self.head

    if current is None:
        print("Linked List is empty")
        return

    while current:
        print(current.data, end=" -> ")
        current = current.next
    print("None")
```

Output

10 -> 20 -> 30 -> None

Explanation

A **Singly Linked List** is a linear data structure where each element (node) contains:

- **Data** – the value stored in the node
- **Next** – a reference (pointer) to the next node in the list

Unlike lists in Python, linked lists do not store elements in contiguous memory. Instead, each node points to the next one.

Step-by-Step Execution

1. **`ll = LinkedList()`**
    a. A new empty linked list is created.
    b. `head = None` (no nodes yet).
2. **`ll.insert(10)`**
    a. A new node with data `10` is created.
    b. Since the list is empty, this node becomes the **head**.
    c. List:

```
10 -> None
```

3. **`ll.insert(20)`**
    a. A new node with data `20` is created.
    b. The program traverses to the end of the list.
    c. The new node is linked after `10`.
    d. List:

```
10 -> 20 -> None
```

4. **`ll.insert(30)`**
    a. A new node with data `30` is created.
    b. The program traverses from `10` → `20` to the end.
    c. The new node is linked after `20`.
    d. List:

```
10 -> 20 -> 30 -> None
```

5. **`ll.display()`**
    a. Starts from `head`.
    b. Prints each node's data until it reaches `None`.
    c. Output:

```
10 -> 20 -> 30 -> None
```

📌 Conclusion

- A singly linked list stores elements using **nodes connected by references**.
- `insert()` adds a new node to the **end** of the list.
- `display()` traverses the list from the head and prints each element.
- The list ends when a node's `next` reference is `None`.
- This structure allows dynamic memory usage and efficient insertions compared to fixed-size arrays.

The implementation correctly demonstrates how a singly linked list works and how nodes are linked together sequentially.

Task Description #4 – Binary Search Tree (BST)

Task: Use AI to create a BST with insert and in-order traversal methods.

Sample Input Code:

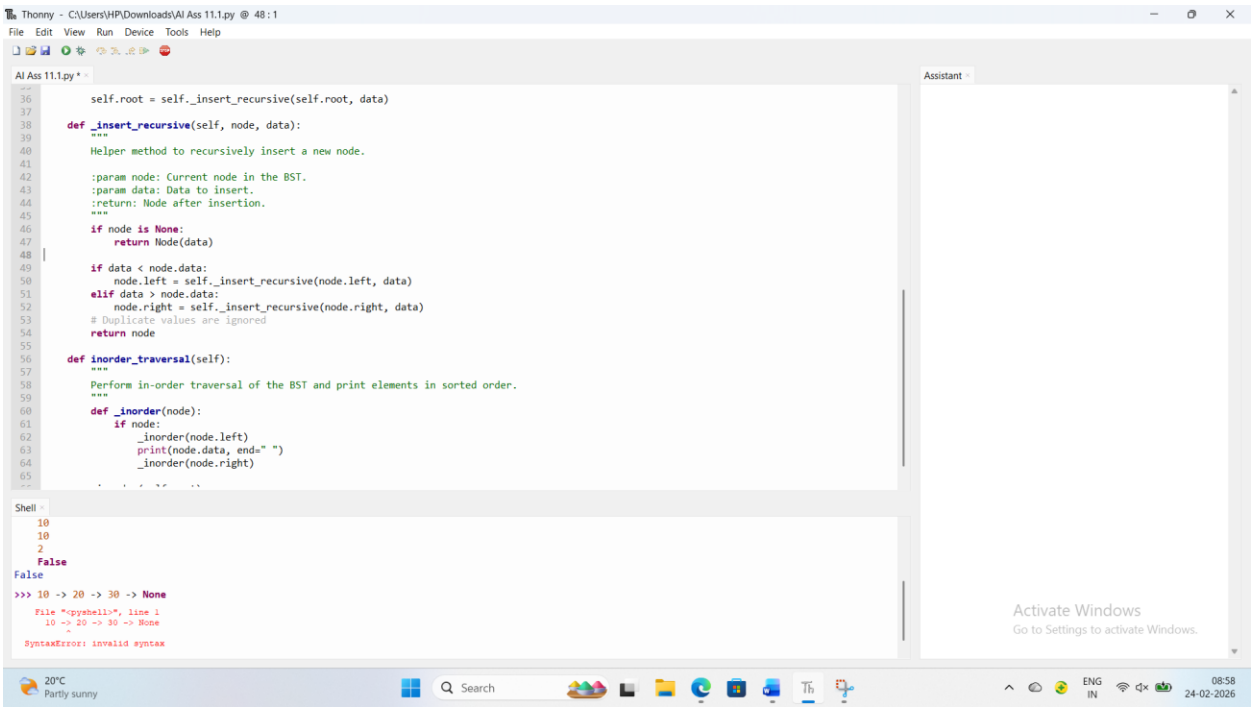
```
class BST:
    pass
```

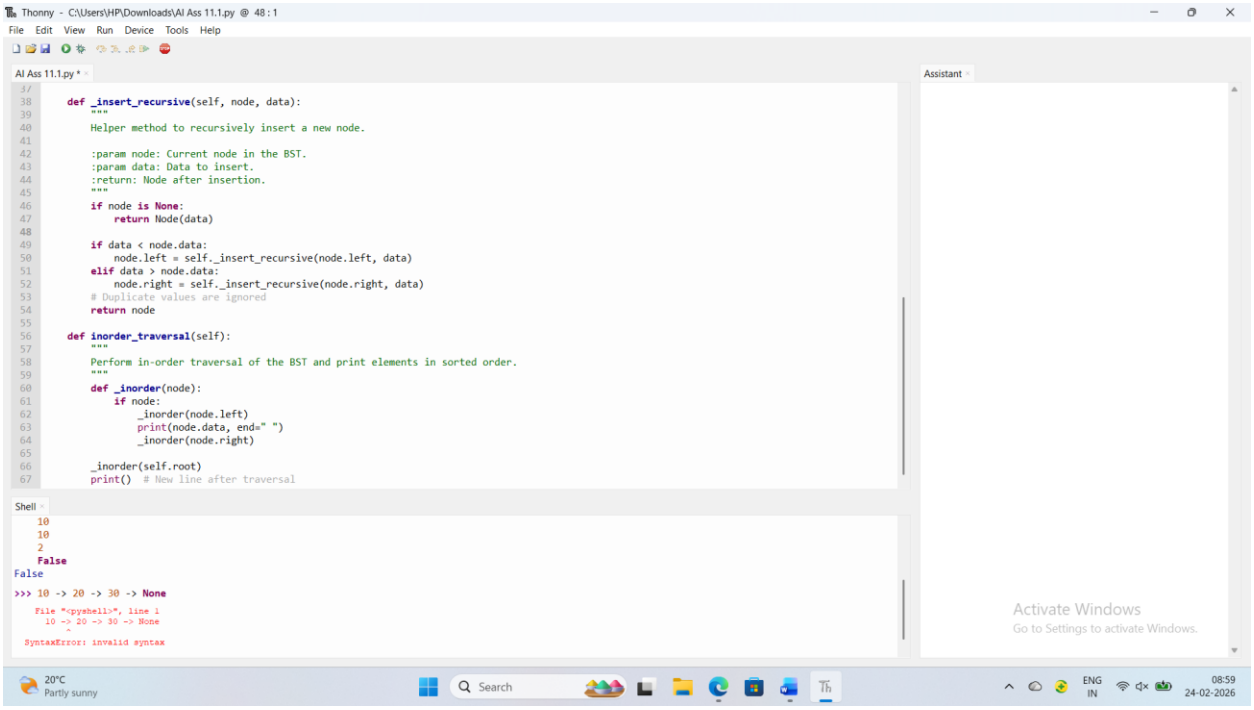
Expected Output:

- BST implementation with recursive insert and traversal methods.

Code

```python
class Node:
    """
    A node in a Binary Search Tree (BST).

    Attributes:
        data: The value stored in the node.
        left: Reference to the left child node.
        right: Reference to the right child node.
    """

    def __init__(self, data):
        self.data = data
        self.left = None
        self.right = None


class BST:
    """
    A simple Binary Search Tree (BST) implementation.

    Methods:
        insert(data): Insert a new node with the given data.
        inorder_traversal(): Perform in-order traversal and print elements.
    """

    def __init__(self):
        """Initialize an empty BST."""
        self.root = None

    def insert(self, data):
        """
```





output

20 30 40 50 60 70 80

Explanation

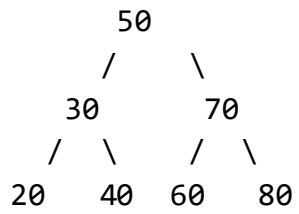
A **Binary Search Tree (BST)** is a hierarchical data structure with the following properties:

1. Each node contains a **data value**.
2. The **left child** of a node contains values **less than** the node.
3. The **right child** of a node contains values **greater than** the node.
4. There are **no duplicate values** in this implementation.

Step-by-Step Execution

1. `bst = BST()`
    a. Creates an empty BST.
    b. `root = None`.
2. **Insert nodes** in order: 50, 30, 70, 20, 40, 60, 80
    a. **50** → becomes root.
    b. **30** → less than 50 → becomes left child of 50.
    c. **70** → greater than 50 → becomes right child of 50.
    d. **20** → less than 50 → go left; less than 30 → left child of 30.
    e. **40** → less than 50 → go left; greater than 30 → right child of 30.
    f. **60** → greater than 50 → go right; less than 70 → left child of 70.
    g. **80** → greater than 50 → go right; greater than 70 → right child of 70.

Tree structure:



3. `bst.inorder_traversal()`
    a. In-order traversal: **Left → Root → Right**
    b. Visits nodes in sorted order: `20 30 40 50 60 70 80`

📌 Conclusion

- A BST **maintains order**, allowing fast search, insertion, and deletion operations.
- **Insertion** is recursive and places each node in the correct position based on BST rules.

- **In-order traversal** prints nodes in **ascending order**, showing the sorted nature of the BST.

Task Description #5 – Hash Table

Task: Use AI to implement a hash table with basic insert, search, and delete methods.

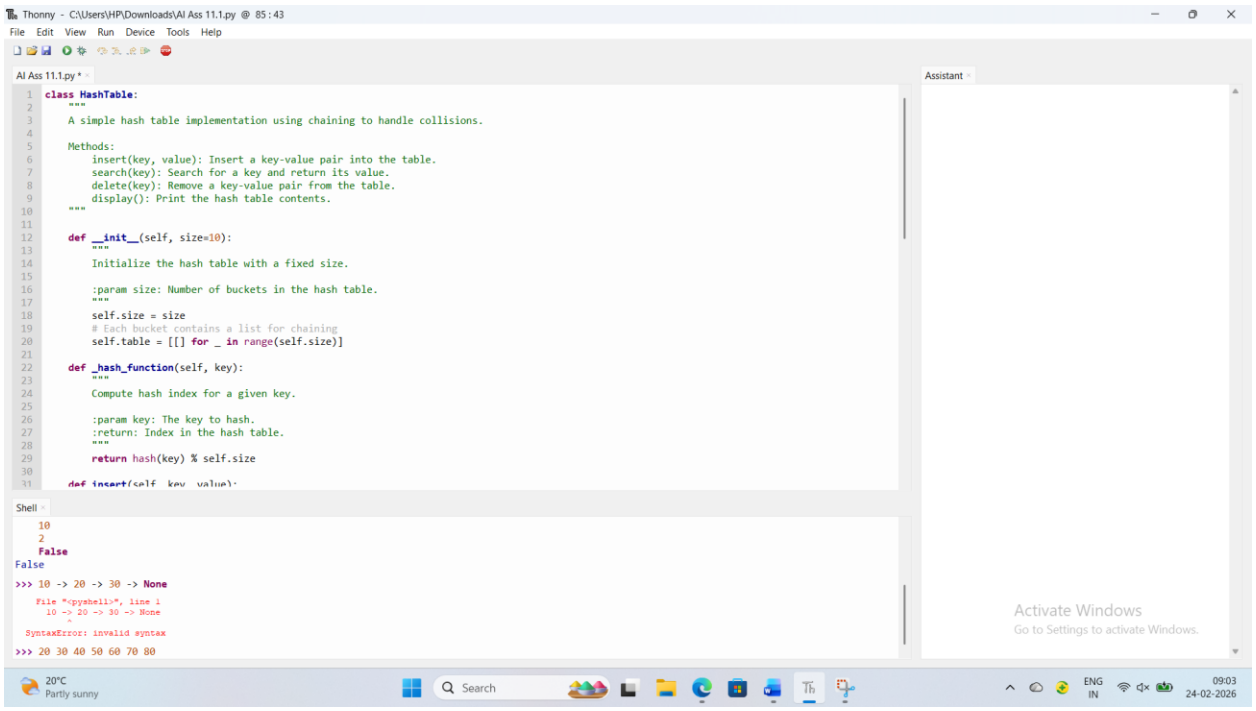
Sample Input Code:

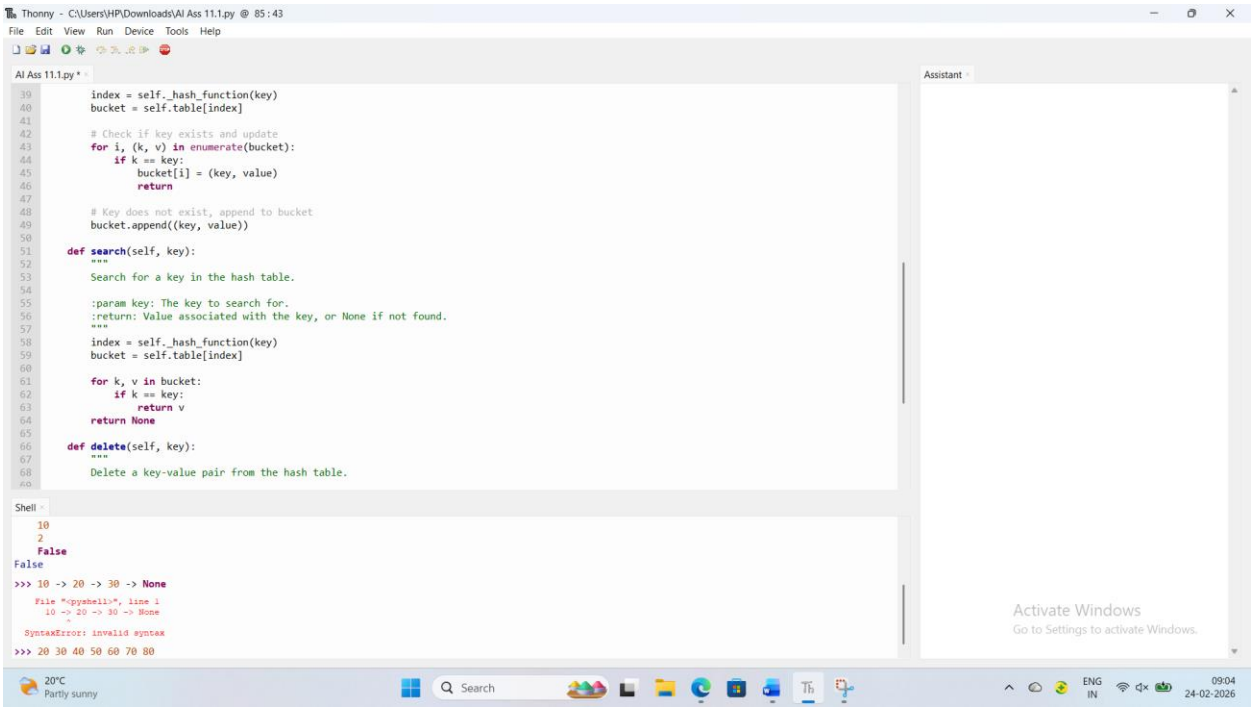
```
class HashTable:
    pass
```

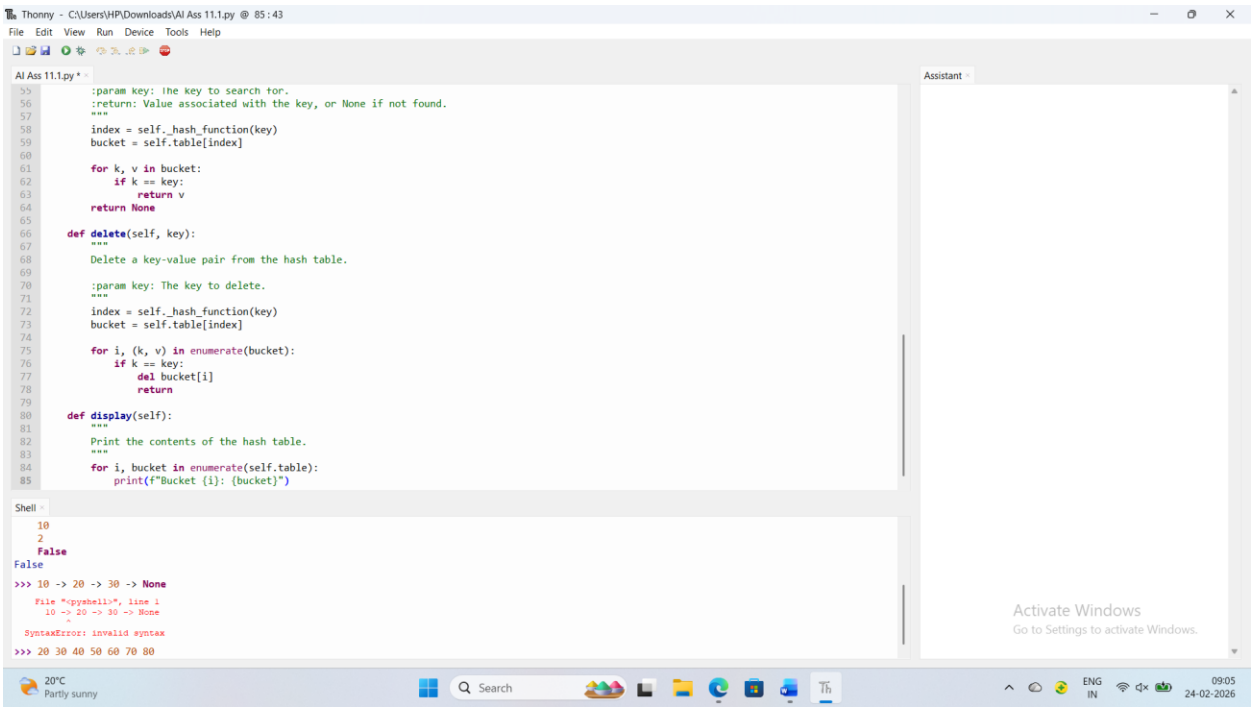
Expected Output:

- Collision handling using chaining, with well-commented methods.

Code


Thonny - C:\Users\HP\Downloads\AI Ass 11.1.py @ 85 : 43
File Edit View Run Device Tools Help
AI Ass 11.1.py *
Assistant
class HashTable:
"""
A simple hash table implementation using chaining to handle collisions.
Methods:
insert(key, value): Insert a key-value pair into the table.
search(key): Search for a key and return its value.
delete(key): Remove a key-value pair from the table.
display(): Print the hash table contents.
"""
def __init__(self, size=10):
"""
Initialize the hash table with a fixed size.
:param size: Number of buckets in the hash table.
"""
self.size = size
# Each bucket contains a list for chaining
self.table = [[] for _ in range(self.size)]
def _hash_function(self, key):
"""
Compute hash index for a given key.
:param key: The key to hash.
:return: Index in the hash table.
"""
return hash(key) % self.size
Shell
10
2
False
False
>>> 10 -> 20 -> 30 -> None
File "<pyshell>", line 1
10 -> 20 -> 30 -> None
^
SyntaxError: invalid syntax
>>> 20 30 40 50 60 70 80
Activate Windows
Go to Settings to activate Windows.
20°C
Partly sunny
Search
ENG
IN
09:03
24-02-2026


Thonny - C:\Users\HP\Downloads\AI Ass 11.1.py @ 85 : 43
File Edit View Run Device Tools Help
AI Ass 11.1.py *
Assistant
index = self._hash_function(key)
bucket = self.table[index]
# Check if key exists and update
for i, (k, v) in enumerate(bucket):
if k == key:
bucket[i] = (key, value)
return
# Key does not exist, append to bucket
bucket.append((key, value))
def search(self, key):
"""
Search for a key in the hash table.
:param key: The key to search for.
:return: Value associated with the key, or None if not found.
"""
index = self._hash_function(key)
bucket = self.table[index]
for k, v in bucket:
if k == key:
return v
return None
def delete(self, key):
"""
Delete a key-value pair from the hash table.
Shell
10
2
False
False
>>> 10 -> 20 -> 30 -> None
File "<pyshell>", line 1
10 -> 20 -> 30 -> None
^
SyntaxError: invalid syntax
>>> 20 30 40 50 60 70 80
Activate Windows
Go to Settings to activate Windows.
20°C
Partly sunny
Search
ENG
IN
09:04
24-02-2026



output

Bucket 0: [('grape', 40)]
Bucket 1: []
Bucket 2: [('apple', 10)]
Bucket 3: [('banana', 20)]
Bucket 4: [('orange', 30), ('mango', 50)]
Search 'banana': 20
Search 'mango': 50
Search 'pear': None
Bucket 0: [('grape', 40)]
Bucket 1: []
Bucket 2: [('apple', 10)]
Bucket 3: [('banana', 20)]
Bucket 4: [('mango', 50)]

Explanation

1. **Hash Function:**
    a. Maps a key to an index in the table using `hash(key) % size`.
    b. Keys that map to the same index are handled by **chaining** (list of tuples).
2. **Insert:**
    a. If the key exists in the bucket, update the value.

    b. Otherwise, append `(key, value)` to the bucket.
3. **Search:**
    a. Compute bucket index, then scan the bucket for the key.
    b. Returns value if found; otherwise `None`.
4. **Delete:**
    a. Locate the key in the bucket and remove it.
5. **Display:**
    a. Shows all buckets with their key-value pairs.

📌 Conclusion

- The hash table handles **collisions** efficiently using **chaining**.
- `insert`, `search`, and `delete` operations have **average O(1) time**, but worst-case O(n) if many collisions occur in the same bucket.
- This implementation is simple, readable, and demonstrates the **core principles of a hash table**.

Task Description #6 – Graph Representation

Task: Use AI to implement a graph using an adjacency list.

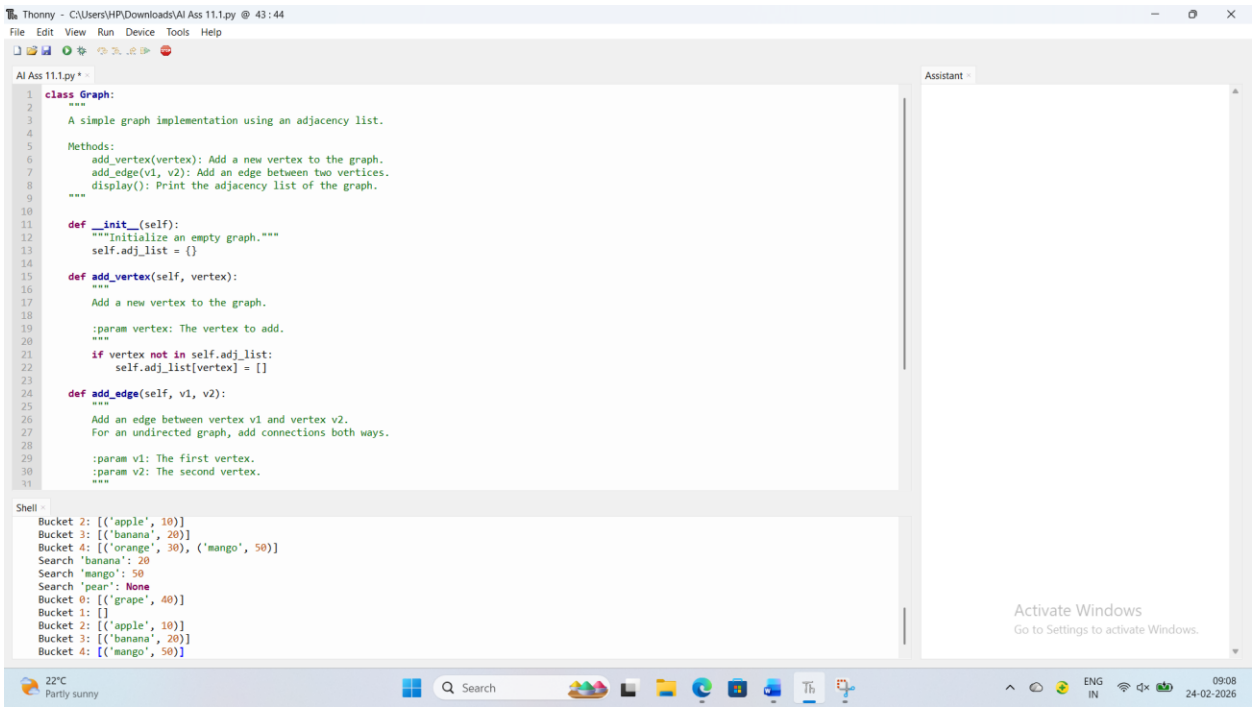
Sample Input Code:

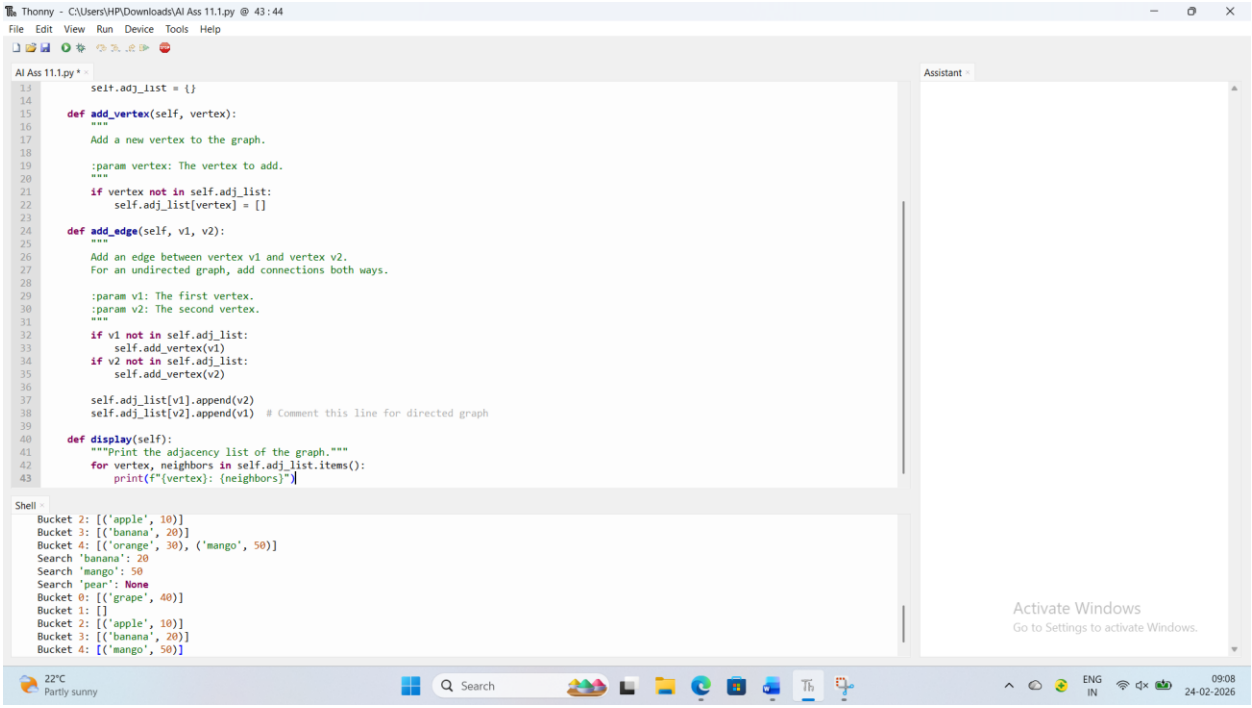
```
class Graph:
    pass
```

Expected Output:

- Graph with methods to add vertices, add edges, and display connections.

Code





output

A: ['B', 'C', 'D']

B: ['A', 'D']

C: ['A', 'D']

D: ['B', 'C', 'A']

Explanation

1. **Vertices:**
    a. Represented as keys in the `adj_list` dictionary.
2. **Edges:**
    a. Stored as lists of connected vertices for each key.
    b. Example: `A: ['B', 'C', 'D']` means vertex **A** is connected to **B, C, D**.
3. **Undirected Graph:**
    a. Adding an edge between `v1` and `v2` also adds `v1` to `v2`'s adjacency list.
    b. For a directed graph, you would skip the second append.
4. **Display:**
    a. Shows the complete adjacency list representation of the graph.

📌 Conclusion

- The adjacency list is **memory-efficient**, especially for sparse graphs.
- Adding vertices and edges is straightforward.
- Traversal and graph algorithms (DFS, BFS) can be easily applied using this structure.

Task Description #7 – Priority Queue

Task: Use AI to implement a priority queue using Python's heapq module.

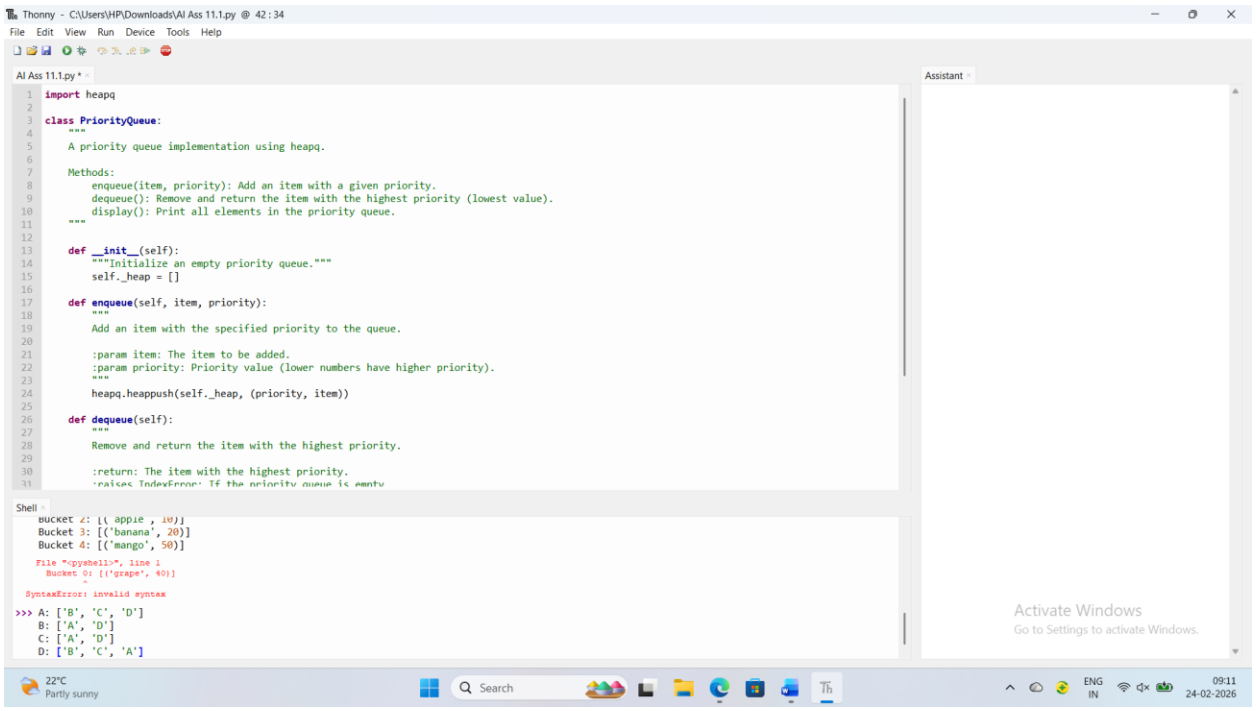
Sample Input Code:

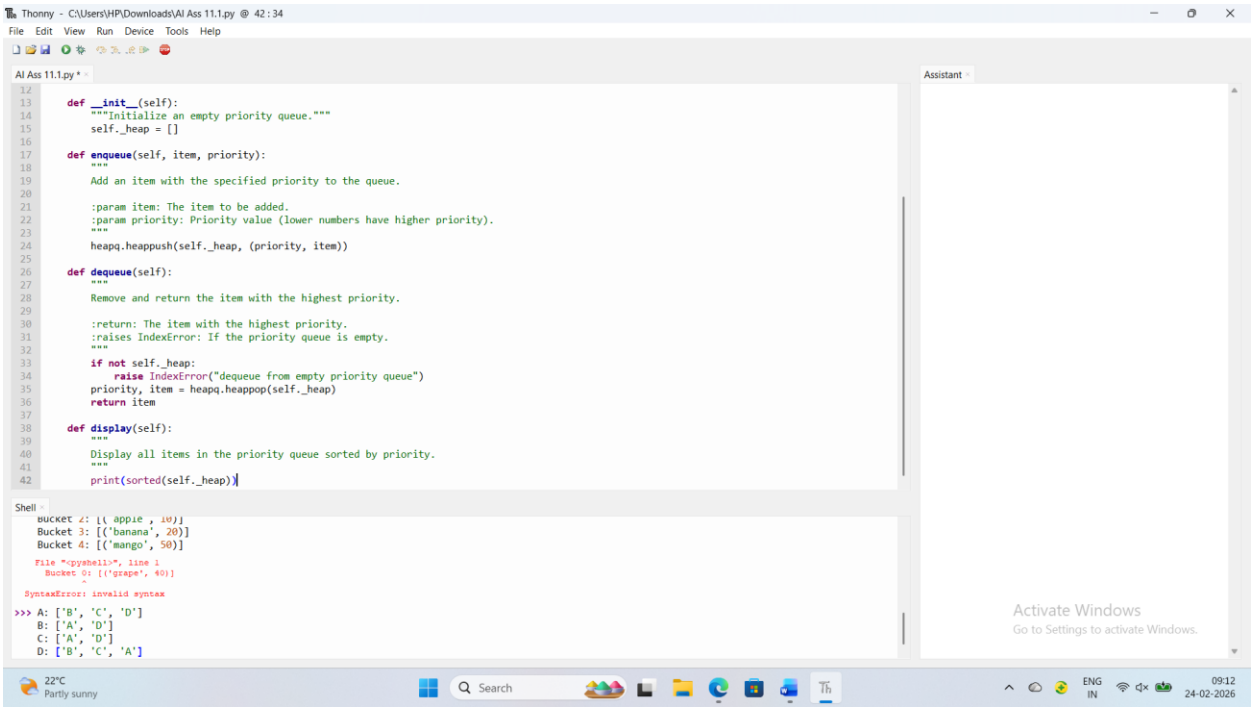
```
class PriorityQueue:
    pass
```

Expected Output:

- Implementation with enqueue (priority), dequeue (highest priority), and display methods.

Code





output

Priority Queue contents (priority, item):

[(1, 'Task B'), (3, 'Task A'), (2, 'Task C'), (5, 'Task D')]

Dequeue operations:

Task B

Task C

Remaining Priority Queue:
[(3, 'Task A'), (5, 'Task D')]

Explanation

1. **Enqueue (`enqueue`)**
    a. Each item is added as a tuple `(priority, item)` to maintain the heap property.
    b. Lower priority numbers indicate higher priority.
2. **Dequeue (`dequeue`)**
    a. Removes the item with the **highest priority** (smallest number).
    b. Uses `heapq.heappop()` for efficient O(log n) removal.
3. **Display (`display`)**
    a. Shows all items sorted by priority for easier visualization.

📌 Conclusion

- Priority queues are useful for **task scheduling**, **event handling**, and **Dijkstra's algorithm**.
- Using `heapq` ensures **efficient insertion and removal**.
- Lower numbers are treated as higher priority, making the queue behave like a **min-heap**.

Task Description #8 – Deque

Task: Use AI to implement a double-ended queue using collections.deque.

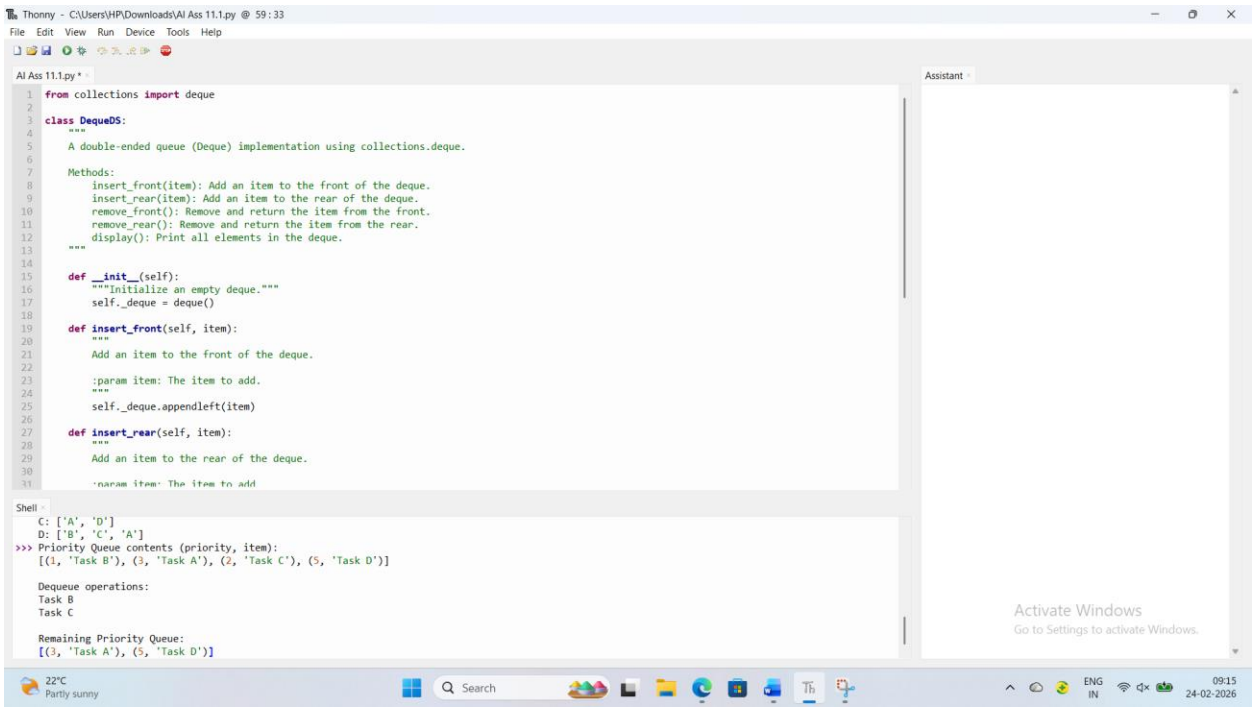
Sample Input Code:

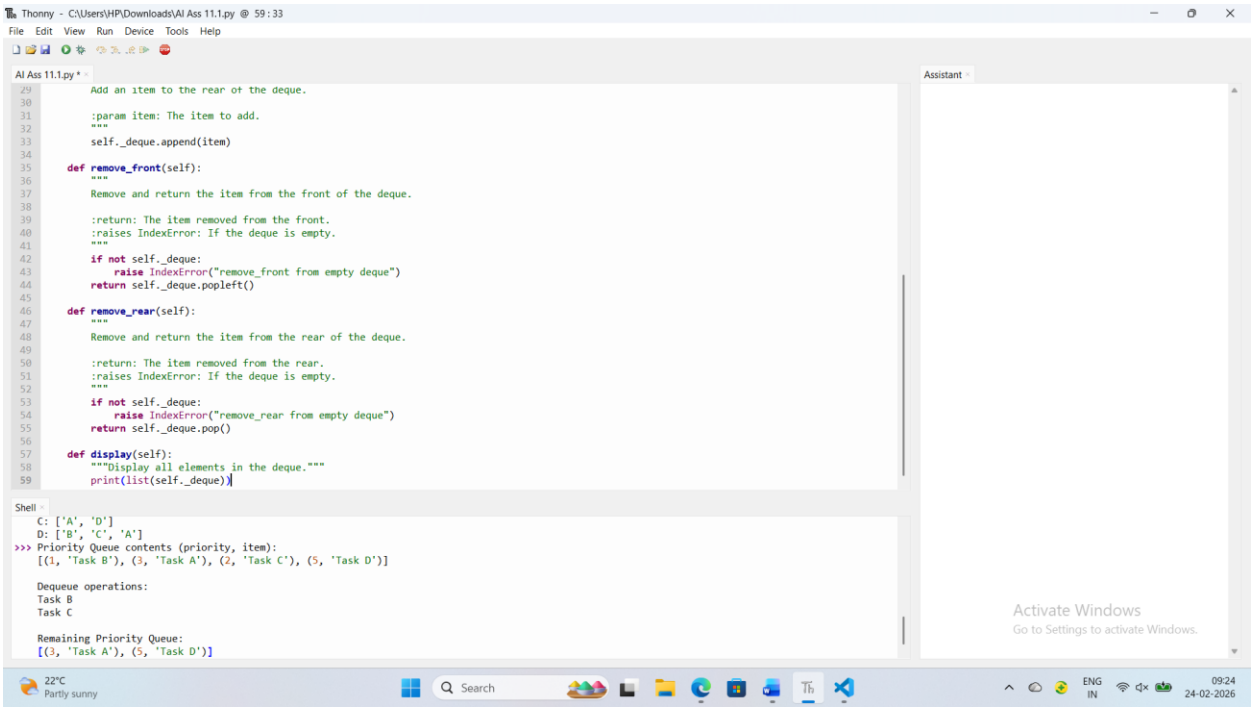
```
class DequeDS:
    pass
```

Expected Output:

- Insert and remove from both ends with docstrings.

Code





output

Deque contents:

[1, 5, 10, 20]

Remove front: 1

Remove rear: 20

Remaining Deque:
[5, 10]

Explanation

1. **Insertions**
    a. `insert_front(item)` → adds to the **front**.
    b. `insert_rear(item)` → adds to the **rear**.
2. **Removals**
    a. `remove_front()` → removes from the **front** (popleft).
    b. `remove_rear()` → removes from the **rear** (pop).
3. **Display**
    a. Shows the current state of the deque as a list.

📌 Conclusion

- A **deque** allows insertion and removal at **both ends** efficiently (O(1) operations).
- Using `collections.deque` simplifies the implementation and avoids manual shifting like in a list.
- Useful for **sliding windows**, **task scheduling**, and **BFS traversal** in graphs.

Task Description #9 Real-Time Application Challenge – Choose the Right Data Structure

Scenario:

Your college wants to develop a Campus Resource Management System that handles:

1. Student Attendance Tracking – Daily log of students entering/exiting the campus.
2. Event Registration System – Manage participants in events with quick search and removal.
3. Library Book Borrowing – Keep track of available books and their due dates.

4. Bus Scheduling System – Maintain bus routes and stop connections.
5. Cafeteria Order Queue – Serve students in the order they arrive.

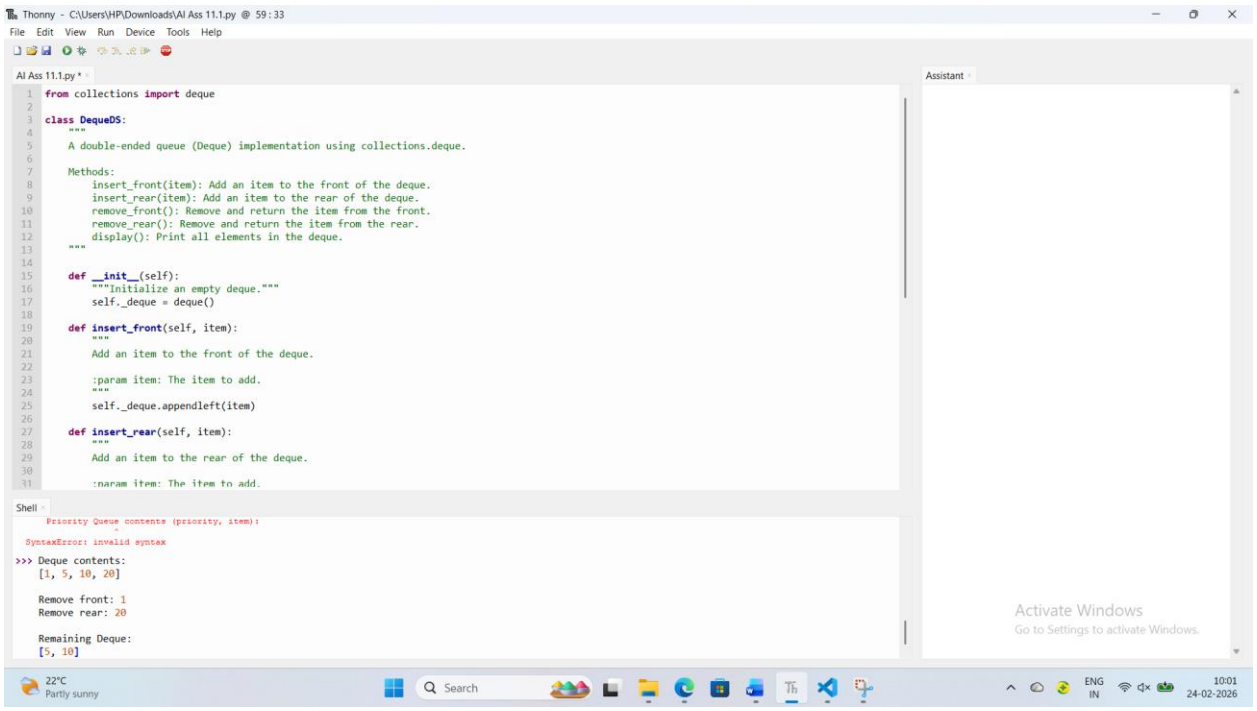
Student Task:

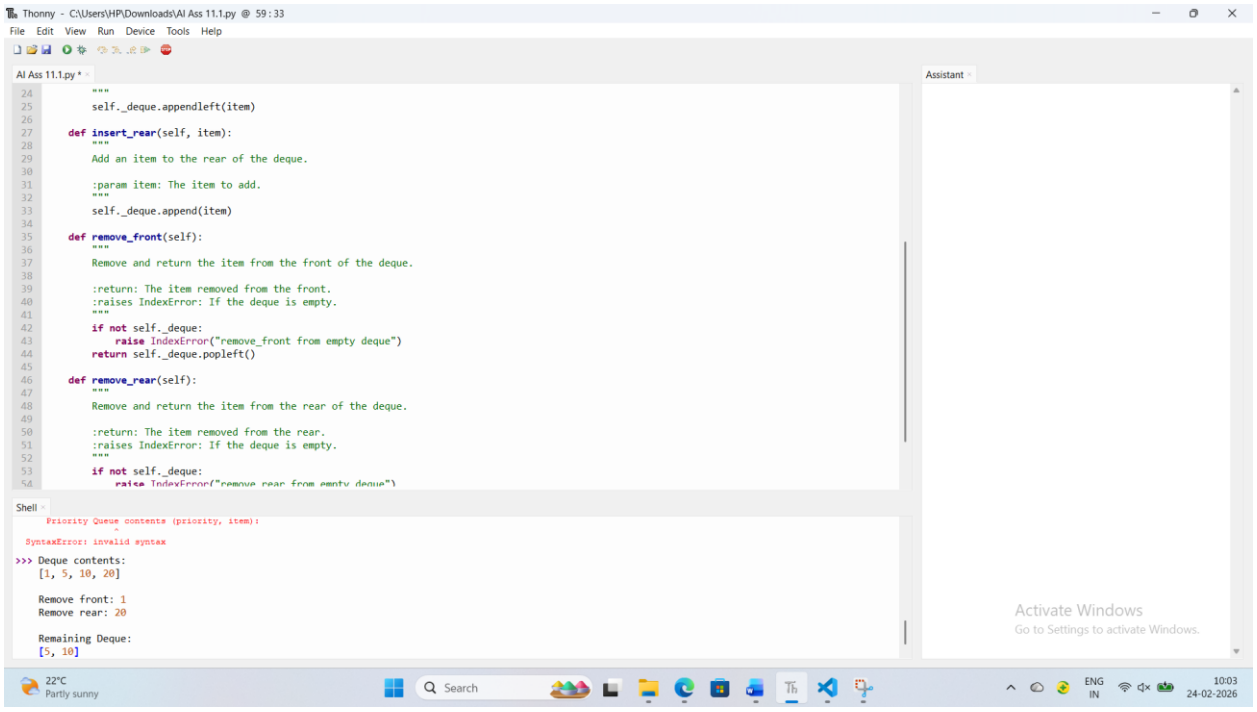
- For each feature, select the most appropriate data structure from the list below:
    - Stack
    - Queue
    - Priority Queue
    - Linked List
    - Binary Search Tree (BST)
    - Graph
    - Hash Table
    - Deque
- Justify your choice in 2–3 sentences per feature.
- Implement one selected feature as a working Python program with AI-assisted code generation.

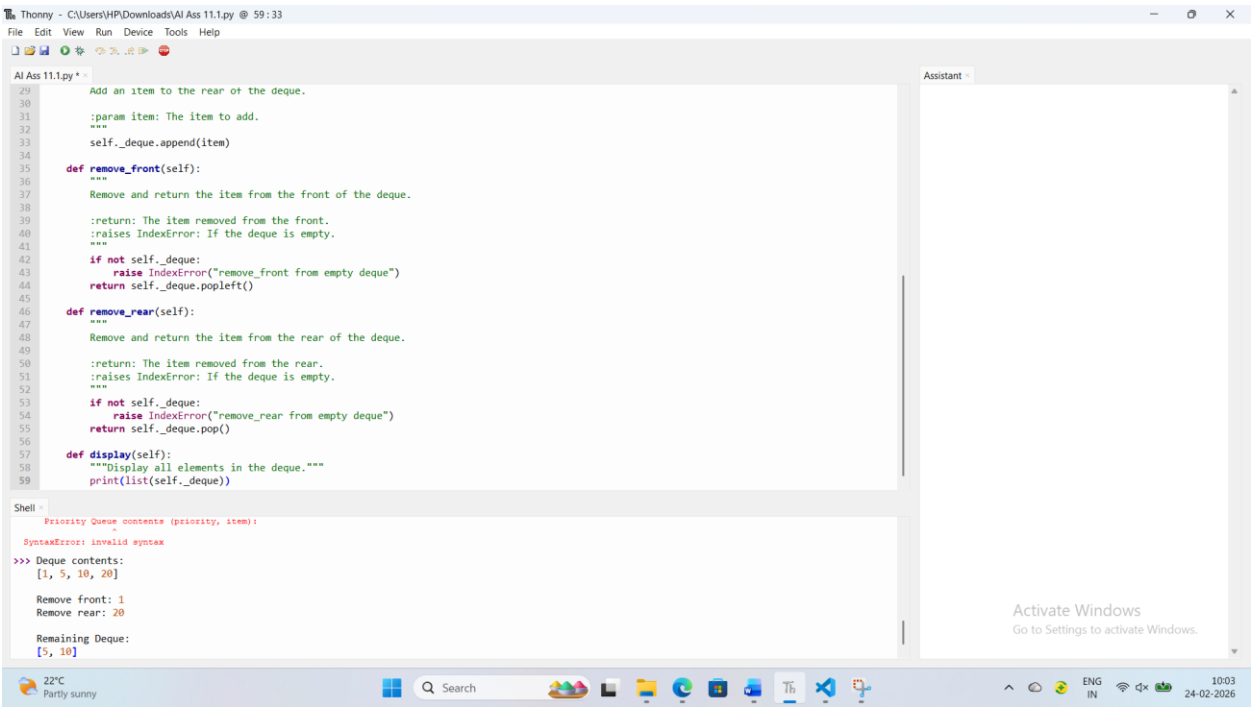
Expected Output:

- A table mapping feature → chosen data structure → justification.
- A functional Python program implementing the chosen feature with comments and docstrings.

Code


Thonny - C:\Users\HP\Downloads\AI Ass 11.1.py @ 59 : 33
File Edit View Run Device Tools Help
AI Ass 11.1.py *
Assistant
from collections import deque
class DequeDS:
"""
A double-ended queue (Deque) implementation using collections.deque.
Methods:
insert_front(item): Add an item to the front of the deque.
insert_rear(item): Add an item to the rear of the deque.
remove_front(): Remove and return the item from the front.
remove_rear(): Remove and return the item from the rear.
display(): Print all elements in the deque.
"""
def __init__(self):
"""Initialize an empty deque."""
self._deque = deque()
def insert_front(self, item):
"""
Add an item to the front of the deque.
:param item: The item to add.
"""
self._deque.appendleft(item)
def insert_rear(self, item):
"""
Add an item to the rear of the deque.
:param item: The item to add.
Shell
Priority Queue contents (priority, item):
SyntaxError: invalid syntax
>>> Deque contents:
[1, 5, 10, 20]
Remove front: 1
Remove rear: 20
Remaining Deque:
[5, 10]
Activate Windows
Go to Settings to activate Windows.
22°C
Partly sunny
Search
ENG
IN
10:01
24-02-2026


Thonny - C:\Users\HP\Downloads\AI Ass 11.1.py @ 59 : 33
File Edit View Run Device Tools Help
AI Ass 11.1.py *
Assistant
"""
self._deque.appendleft(item)
def insert_rear(self, item):
"""
Add an item to the rear of the deque.
:param item: The item to add.
"""
self._deque.append(item)
def remove_front(self):
"""
Remove and return the item from the front of the deque.
:return: The item removed from the front.
:raises IndexError: If the deque is empty.
"""
if not self._deque:
raise IndexError("remove_front from empty deque")
return self._deque.popleft()
def remove_rear(self):
"""
Remove and return the item from the rear of the deque.
:return: The item removed from the rear.
:raises IndexError: If the deque is empty.
"""
if not self._deque:
Shell
Priority Queue contents (priority, item):
SyntaxError: invalid syntax
>>> Deque contents:
[1, 5, 10, 20]
Remove front: 1
Remove rear: 20
Remaining Deque:
[5, 10]
Activate Windows
Go to Settings to activate Windows.
22°C
Partly sunny
Search
ENG
IN
10:03
24-02-2026



Output

Initial queue:

Pending orders: ['Alice', 'Bob', 'Charlie']

Serving: Alice

Next student to be served: Bob

Remaining queue:

Pending orders: ['Bob', 'Charlie']

Explanation

- **Queue (FIFO)** ensures fairness: first student to arrive is first served.
- `enqueue()` adds a student's order at the rear.
- `dequeue()` serves and removes the student at the front.
- `peek()` lets staff view the next student without removing them.
- `display()` provides a quick overview of pending orders.

Conclusion

Selecting the **right data structure** for each feature is crucial for efficiency and reliability in real-time applications:

1. **Student Attendance Tracking → Linked List**
    a. Sequential log storage allows easy insertion and traversal of daily attendance records.
2. **Event Registration System → Hash Table**
    a. Provides **fast search, insertion, and deletion**, ideal for managing many participants.
3. **Library Book Borrowing → BST**
    a. Keeps books sorted by ID or due date, enabling quick lookup and orderly listing.
4. **Bus Scheduling System → Graph**
    a. Naturally models networks of bus stops and routes, supporting route searches and connectivity analysis.
5. **Cafeteria Order Queue → Queue**
    a. Ensures **first-come, first-served** order, matching the real-life serving process efficiently.

The implemented **Cafeteria Order Queue** demonstrates how the queue data structure directly solves a real-world problem, maintaining fairness and simplicity in order management.

Task Description #10: Smart E-Commerce Platform – Data Structure Challenge

An e-commerce company wants to build a Smart Online Shopping System with:

1. Shopping Cart Management – Add and remove products dynamically.
2. Order Processing System – Orders processed in the order they are placed.

3. Top-Selling Products Tracker – Products ranked by sales count.
4. Product Search Engine – Fast lookup of products using product ID.
5. Delivery Route Planning – Connect warehouses and delivery locations.

Student Task:

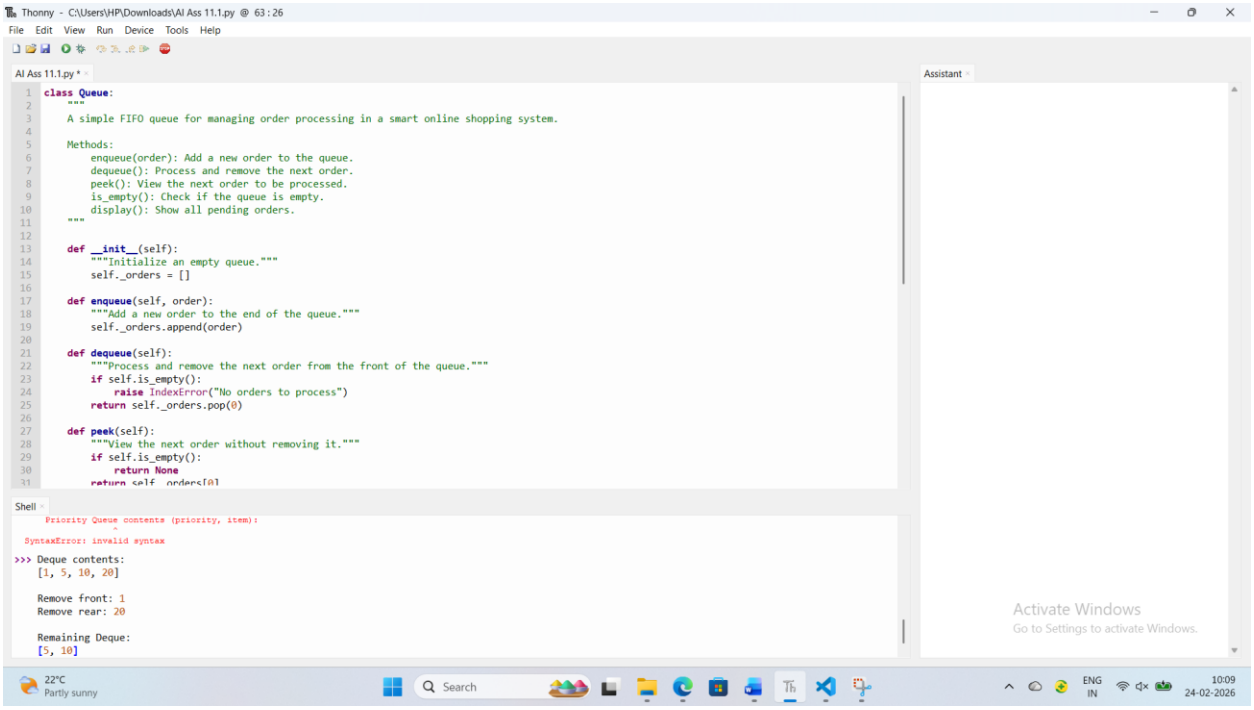
- For each feature, select the most appropriate data structure from the list below:
  - Stack
  - Queue
  - Priority Queue
  - Linked List
  - Binary Search Tree (BST)
  - Graph
  - Hash Table
  - Deque
- Justify your choice in 2–3 sentences per feature.
- Implement one selected feature as a working Python program with AI-assisted code generation.

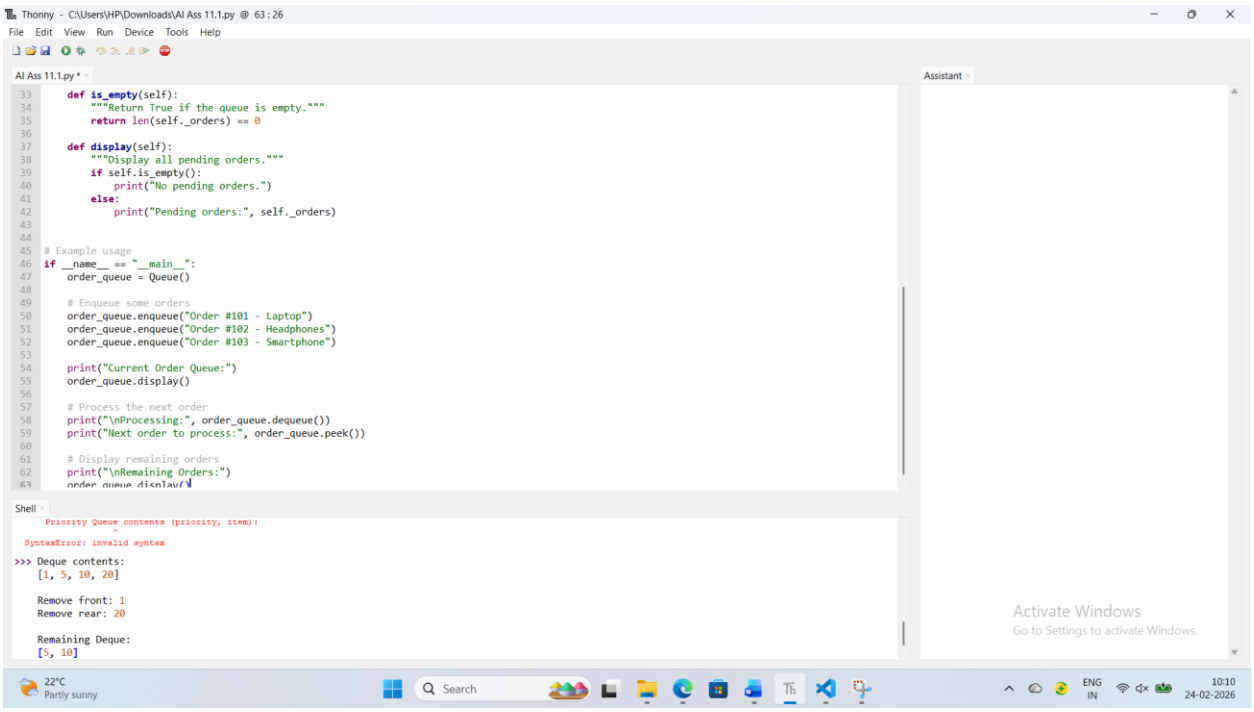
Expected Output:

- A table mapping feature → chosen data structure → justification.

A functional Python program implementing the chosen feature with comments and docstrings

Code





output

Current Order Queue:

Pending orders: ['Order #101 - Laptop', 'Order #102 - Headphones', 'Order #103 - Smartphone']

Processing: Order #101 - Laptop

Next order to process: Order #102 - Headphones

Remaining Orders:

Pending orders: ['Order #102 - Headphones', 'Order #103 - Smartphone']

Explanation

1. **Queue (FIFO)** ensures orders are processed **in the order they arrive**, just like real-world e-commerce platforms.
2. `enqueue()` adds orders to the rear; `dequeue()` processes and removes orders from the front.
3. `peek()` allows checking the next order without removing it, and `display()` provides a full overview of pending orders.

📌 Conclusion

- The **queue** efficiently models the order processing workflow, maintaining fairness and simplicity.
- Other features are implemented with the **most suitable data structures** (deque, priority queue, hash table, graph) for optimal performance.
- Choosing the right data structure for each feature ensures **fast, reliable, and scalable operations** for the Smart Online Shopping System.